1 Знакомьтесь: Kafka

Деятельность любого предприятия питается данными. Мы получаем информацию, анализируем ее, выполняем над ней какие-либо действия и создаем новые данные в качестве результатов. Все приложения создают данные — журнальные сообщения, показатели, информацию об операциях пользователей, исходящие сообщения или что-то еще. Каждый байт данных что-нибудь да значит — что-нибудь, определяющее дальнейшие действия. А чтобы понять, что именно, нам нужно переместить данные из места создания туда, где их можно проанализировать. Это мы каждый день наблюдаем на таких сайтах, как Amazon, где щелчки мышью на интересующих нас товарах превращаются в демонстрируемые нам же чуть позже рекомендации.

От скорости этого процесса зависят адаптивность и быстрота реакции нашего предприятия. Чем меньше усилий мы тратим на перемещение данных, тем больше можем уделить внимания основной деятельности. Именно поэтому конвейер — ключевой компонент в ориентированном на работу с данными предприятии. Способ перемещения данных оказывается практически столь же важен, как и сами данные.

> Первопричиной всякого спора ученых является нехватка данных. Постепенно мы приходим к согласию относительно того, какие данные нужны, получаем эти данные, и данные решают проблему. Или я оказываюсь прав, или вы, или мы оба ошибаемся. И можно двигаться дальше.
>
> *Нил Деграсс Тайсон*

Обмен сообщениями по типу «публикация/подписка»

Прежде чем перейти к обсуждению нюансов Apache Kafka, важно разобраться в обмене сообщениями по типу «публикация/подписка» и причине, по которой оно столь важно. *Обмен сообщениями по типу «публикация/подписка»* (publish/subscribe messaging) — паттерн проектирования, отличающийся тем, что отправитель (издатель) элемента данных (сообщения) не направляет его конкретному потребителю. Вместо этого он каким-то образом классифицирует сообщения,

а потребитель (подписчик) подписывается на определенные классы сообщений. В системы типа «публикация/подписка» для упрощения этих действий часто включают брокер — центральный пункт публикации сообщений.

С чего все начинается

Множество сценариев использования публикации/подписки начинается одинаково — с простой очереди сообщений или канала обмена ими между процессами. Например, вы создали приложение, которому необходимо отправлять куда-либо мониторинговую информацию, для чего приходится создавать прямое соединение между вашим приложением и приложением, отображающим показатели в инструментальной панели, и «проталкивать» последние через это соединение (рис. 1.1).

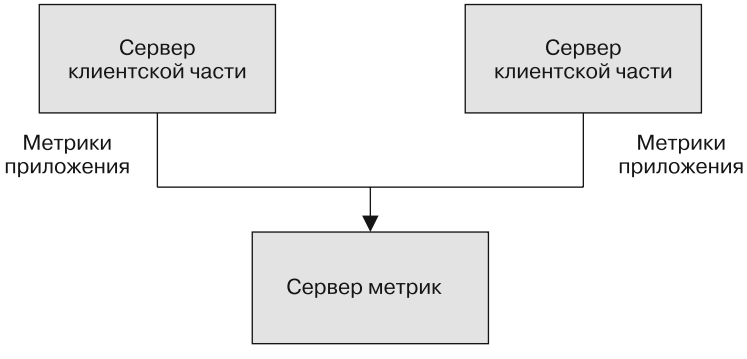

Рис. 1.1. Отдельный непосредственный издатель показателей

Это простое решение простой задачи, удобное для начала мониторинга. Но вскоре вам захочется анализировать показатели за больший период времени, а в инструментальной панели это не слишком удобно. Вы создадите новый сервис для получения показателей, их хранения и анализа. Для этого измените свое приложение так, чтобы оно могло записывать их в обе системы. К тому времени у вас появится еще три генерирующих показатели приложения, каждое из которых будет точно так же подключаться к этим двум сервисам. Один из коллег предложит идею активных опросов сервисов для оповещения, так что вы добавите к каждому из приложений сервер, выдающий показатели по запросу. Вскоре у вас появятся дополнительные приложения, использующие эти серверы для получения отдельных показателей в различных целях. Архитектура станет напоминать рис. 1.2, возможно, соединениями, которые еще труднее отслеживать.

Некоторая недоработка тут очевидна, так что вы решаете ее исправить. Создаете единое приложение, получающее показатели от всех имеющихся приложений и включающее сервер, предназначенный для запроса этих показателей для всех систем, которым они нужны. В результате сложность архитектуры снижается (рис. 1.3). Поздравляем, вы создали систему обмена сообщениями по типу «публикация/подписка»!

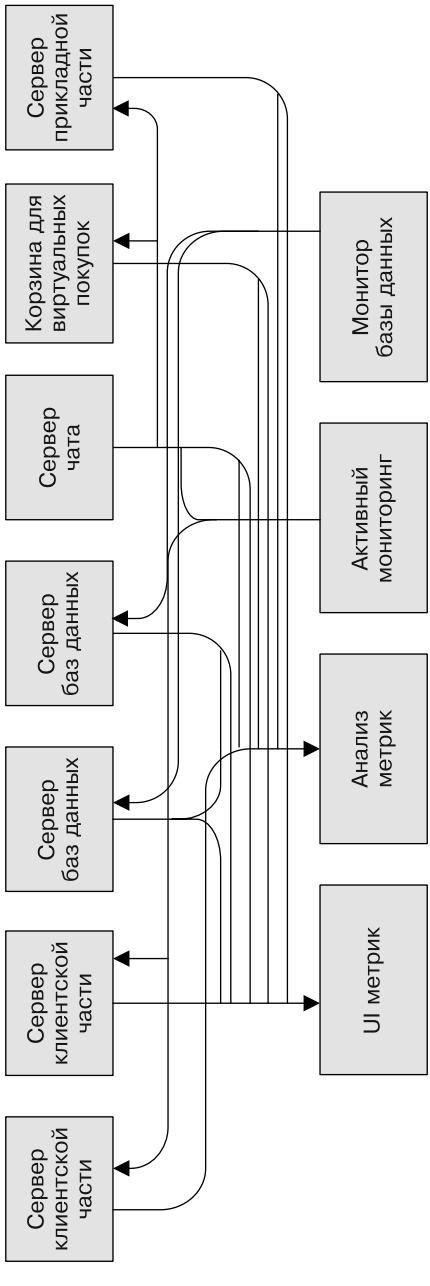

Рис. 1.2. Множество издателей показателей, использующих прямые соединения

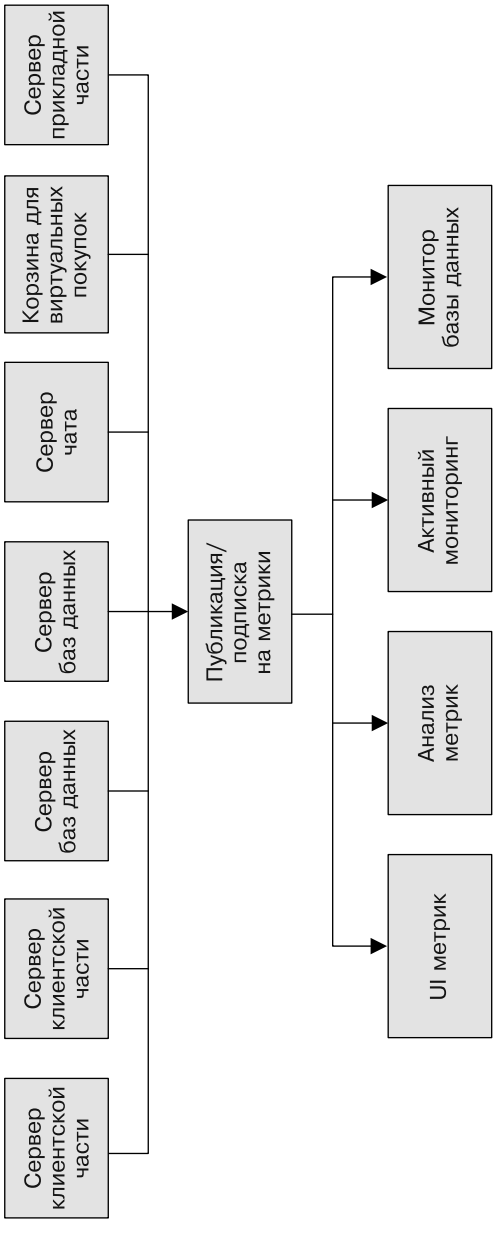

Рис. 1.3. Система публикации/подписки на показатели

Отдельные системы организации очередей

В то время как вы боролись с показателями, один из ваших коллег аналогичным образом трудился над журнальными сообщениями. А еще один работал над отслеживанием действий пользователей на веб-сайте клиентской части и передачей этой информации разработчикам, занимающимся машинным обучением, параллельно с формированием отчетов для начальства. Вы все шли одним и тем же путем создания систем, расцепляющих издателей информации и подписчиков на нее. Подобная инфраструктура с тремя отдельными системами публикации/подписки показана на рис. 1.4.

Использовать ее намного удобнее, чем прямые соединения (как на рис. 1.2), но возникает существенное дублирование. Компании приходится сопровождать несколько систем организации очередей, в каждой из которых имеются собственные ошибки и ограничения. А между тем вы знаете, что скоро появятся новые сценарии использования обмена сообщениями. Необходима единая централизованная система, поддерживающая публикацию обобщенных типов данных, которая могла бы развиваться по мере расширения вашего бизнеса.

Открываем для себя систему Kafka

Apache Kafka — система публикации сообщений и подписки на них, предназначенная для решения поставленной задачи. Ее часто называют распределенным журналом фиксации транзакций или, в последнее время, распределенной платформой потоковой обработки. Файловая система или журнал фиксации транзакций базы данных предназначены для обеспечения долговременного хранения всех транзакций таким образом, чтобы можно было их воспроизвести с целью восстановления согласованного состояния системы. Аналогично данные в Kafka хранятся долго, и их можно читать когда угодно. Кроме того, они могут распределяться по системе в качестве меры дополнительной защиты от сбоев, равно как и ради повышения производительности.

Сообщения и пакеты

Используемая в Kafka единица данных называется *сообщением* (message). Если ранее вы работали с базами данных, то можете рассматривать сообщение как аналог *строки* (row) или *записи* (record). С точки зрения Kafka сообщение представляет собой просто массив байтов, так что для нее содержащиеся в нем данные не имеют формата или какого-либо смысла. В сообщении может быть дополнительный элемент метаданных, называемый *ключом* (key). Он также представляет собой массив байтов и, как и сообщение, не несет для Kafka никакого смысла. Ключи используются при необходимости лучше управлять записью сообщений в разделы. Простейшая схема такова: генерация единообразного хеш-значения ключа с по-

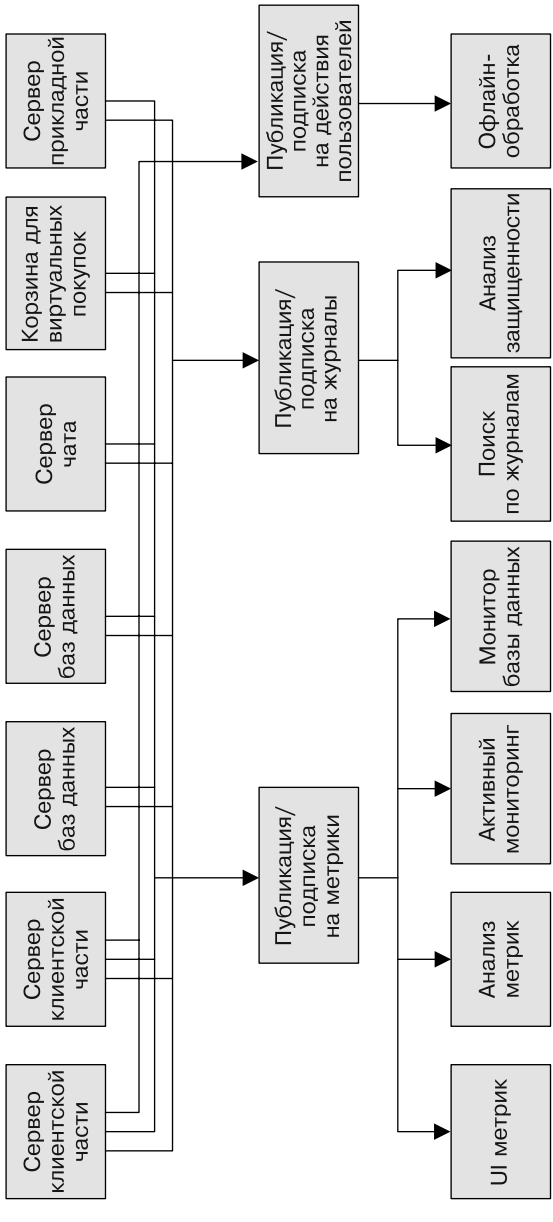

Рис. 1.4. Несколько систем публикации/подписки

следующим выбором номера раздела для сообщения путем деления указанного значения по модулю общего числа разделов в теме. Это гарантирует попадание сообщений с одним ключом в один раздел. Мы обсудим ключи подробнее в главе 3.

Для большей эффективности сообщения в Kafka записываются пакетами. *Пакет* (batch) представляет собой просто набор сообщений, относящихся к одной теме и одному разделу. Передача каждого сообщения туда и обратно по сети привела бы к существенному перерасходу ресурсов, а объединение сообщений в пакет эту проблему уменьшает. Конечно, необходимо соблюдать баланс между временем задержки и пропускной способностью: чем больше пакеты, тем больше сообщений можно обрабатывать за единицу времени, но тем дольше распространяется отдельное сообщение. Пакеты обычно подвергаются сжатию, что позволяет передавать и хранить данные более эффективно за счет некоторого расхода вычислительных ресурсов.

Схемы

Хотя сообщения для Kafka — всего лишь непрозрачные массивы байтов, рекомендуется накладывать на содержимое сообщений дополнительную структуру — схему, которая позволяла бы с легкостью их разбирать. Существует много вариантов задания *схемы* сообщений в зависимости от потребностей конкретного приложения. Упрощенные системы, например, нотация объектов JavaScript (JavaScript Object Notation, JSON) и расширяемый язык разметки (Extensible Markup Language, XML), просты в использовании, их удобно читать человеку. Однако им не хватает таких свойств, как ошибкоустойчивая работа с типами и совместимость разных версий схемы. Многим разработчикам Kafka нравится Apache Avro — фреймворк сериализации, изначально предназначенный для Hadoop. Avro обеспечивает компактный формат сериализации, схемы, отделенные от содержимого сообщений и не требующие генерации кода при изменении, а также сильную типизацию данных и эволюцию схемы с прямой и обратной совместимостью.

Для Kafka важен единообразный формат данных, ведь он дает возможность расцеплять код записи и чтения сообщений. При тесном сцеплении этих задач приходится модифицировать приложения-подписчики, чтобы они могли работать не только со старым, но и с новым форматом данных. Только после этого можно будет использовать новый формат в публикующих сообщения приложениях. Благодаря применению четко заданных схем и хранению их в общем репозитории сообщения в Kafka можно читать, не координируя действия. Мы рассмотрим схемы и сериализацию подробнее в главе 3.

Темы и разделы

Сообщения в Kafka распределяются по *темам* (topics, иногда называют топиками). Ближайшая аналогия — таблица базы данных или каталог файловой системы. Темы в свою очередь разбиваются на *разделы* (partitions). Если вернуться к описанию журнала фиксации, то раздел представляет собой отдельный журнал. Сообщения

записываются в него путем добавления в конец, а читаются от начала к концу. Заметим: поскольку тема обычно состоит из нескольких разделов, нет никаких гарантий упорядоченности сообщений в пределах всей темы — лишь в пределах отдельного раздела. На рис. 1.5 показана тема с четырьмя разделами, в конец каждого из которых добавляются сообщения. Благодаря разделам Kafka обеспечивает также избыточность и масштабируемость. Любой из разделов можно разместить на отдельном сервере, что означает возможность горизонтального масштабирования системы на несколько серверов с целью достижения производительности, далеко выходящей за пределы возможностей одного сервера.

Рис. 1.5. Представление темы с несколькими разделами

При обсуждении данных, находящихся в таких системах, как Kafka, часто используется термин *поток данных* (stream). Чаще всего поток данных считается отдельной темой, независимо от количества разделов представляющей собой отдельный поток данных, перемещающихся от производителей к потребителям. Обычно сообщения рассматривают подобным образом при обсуждении потоковой обработки, при которой фреймворки, в частности Kafka Streams, Apache Samza и Storm, работают с сообщениями в режиме реального времени. Их принцип работы подобен принципу работы офлайн-фреймворков, в частности Hadoop, предназначенных для работы с блоками данных. Обзор темы потоковой обработки приведен в главе 11.

Производители и потребители

Пользователи Kafka делятся на два основных типа: производители (генераторы) и потребители. Существуют также продвинутые клиентские API — API Kafka Connect для интеграции данных и Kafka Streams для потоковой обработки. Продвинутые клиенты используют производители и потребители в качестве строительных блоков, предоставляя на их основе функциональность более высокого уровня.

Производители (producers) генерируют новые сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *издателями* (publishers) или *авторами* (writers). В целом производители сообщений создают их для конкретной темы. По умолчанию производителю не важно, в какой раздел записывается конкретное сообщение, он будет равномерно поставлять сообще-

ния во все разделы темы. В некоторых случаях производитель направляет сообщение в конкретный раздел, для чего обычно служат ключ сообщения и объект `Partitioner`, генерирующий хеш ключа и устанавливающий его соответствие с конкретным разделом. Это гарантирует запись всех сообщений с одинаковым ключом в один и тот же раздел. Производитель может также воспользоваться собственным объектом `Partitioner` со своими бизнес-правилами распределения сообщений по разделам. Более подробно поговорим о производителях в главе 3.

Потребители (consumers) читают сообщения. В других системах обмена сообщениями по типу «публикация/подписка» их называют *подписчиками* (subscribers) или *читателями* (readers). Потребитель подписывается на одну тему или более и читает сообщения в порядке их создания. Он отслеживает, какие сообщения он уже прочитал, запоминая смещение сообщений. *Смещение* (offset) (непрерывно возрастающее целочисленное значение) — еще один элемент метаданных, который Kafka добавляет в каждое сообщение при генерации. Смещения сообщений в конкретном разделе не повторяются. Благодаря сохранению смещения последнего полученного сообщения для каждого раздела в хранилище ZooKeeper или самой Kafka потребитель может приостанавливать и возобновлять свою работу, не забывая, в каком месте он читал.

Потребители работают в составе *групп потребителей* (consumer groups) — одного или нескольких потребителей, объединившихся для обработки темы. Организация в группы гарантирует чтение каждого раздела только одним членом группы. На рис. 1.6 представлены три потребителя, объединенные в одну группу для обработки темы. Два потребителя обрабатывают по одному разделу, а третий — два. Соответствие потребителя разделу иногда называют *принадлежностью* (ownership) раздела данному потребителю.

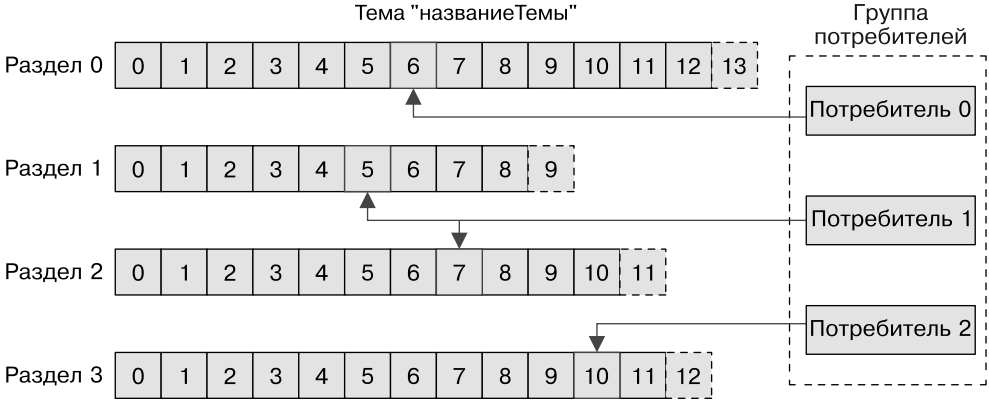

Рис. 1.6. Чтение темы группой потребителей

Таким образом, потребители получают возможность горизонтального масштабирования для чтения темы с большим количеством сообщений. Кроме того, в случае

сбоя отдельного потребителя оставшиеся члены группы перераспределят разделы так, чтобы взять на себя его задачу. Потребители и группы потребителей подробнее описываются в главе 4.

Брокеры и кластеры

Отдельный сервер Kafka называется *брокером* (broker). Брокер получает сообщения от производителей, присваивает им смещения и отправляет их в дисковое хранилище. Он также обслуживает потребители и отвечает на запросы выборки из разделов, возвращая записанные на диск сообщения. В зависимости от конкретного аппаратного обеспечения и его производительности отдельный брокер может с легкостью обрабатывать тысячи разделов и миллионы сообщений в секунду.

Брокеры Kafka предназначены для работы в составе *кластера* (cluster). Один из брокеров кластера функционирует в качестве *контроллера* (cluster controller). Контроллер кластера выбирается автоматически из числа работающих членов кластера. Контроллер отвечает за административные операции, включая распределение разделов по брокерам и мониторинг отказов последних. Каждый раздел принадлежит одному из брокеров кластера, который называется его *ведущим* (leader). Раздел можно назначить нескольким брокерам, в результате чего произойдет ее репликация (рис. 1.7). Это обеспечивает избыточность сообщений в разделе, так что в случае сбоя ведущего другой брокер сможет занять его место. Однако все потребители и производители, работающие в этом разделе, должны соединяться с ведущим. Кластерные операции, включая репликацию разделов, подробно рассмотрены в главе 6.

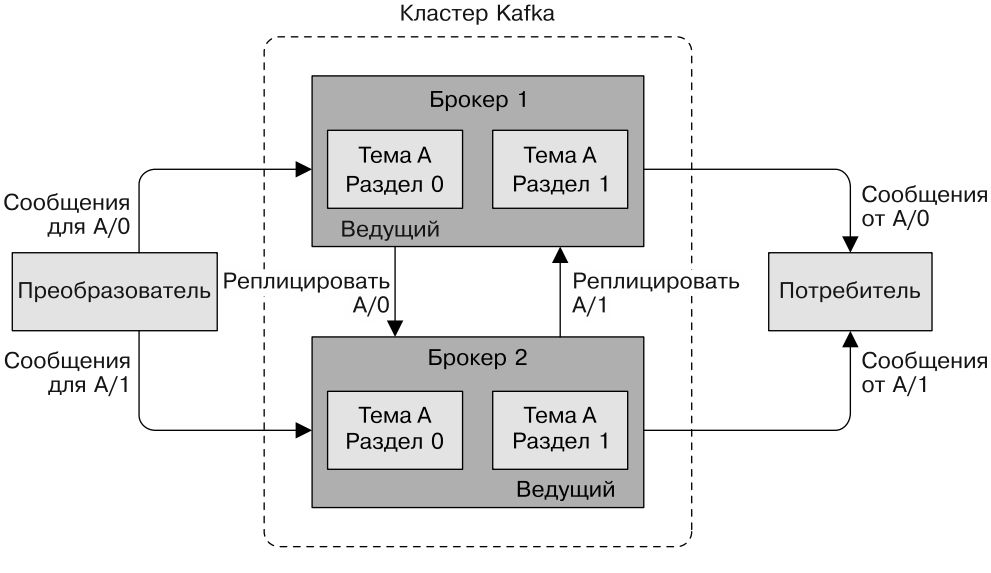

Рис. 1.7. Репликация разделов в кластере

Ключевая возможность Apache Kafka — *сохранение информации* (retention) в течение длительного времени. В настройки брокеров Kafka включается длительность хранения тем по умолчанию — или в течение определенного промежутка времени (например, 7 дней), или до достижения темой определенного размера в байтах (например, 1 Гбайт). Превысившие эти пределы сообщения становятся недействительными и удаляются, так что настройки сохранения соответствуют минимальному количеству доступной в каждый момент информации. Можно задавать настройки сохранения и для отдельных тем, чтобы сообщения хранились только до тех пор, пока они нужны. Например, тема для отслеживания действий пользователей можно хранить несколько дней, в то время как параметры приложений — лишь несколько часов. Можно также настроить для тем вариант хранения *сжатых журналов* (log compacted). При этом Kafka будет хранить лишь последнее сообщение с конкретным ключом. Это может пригодиться для таких данных, как журналы изменений, в случае, когда нас интересует только последнее изменение.

Несколько кластеров

По мере роста развертываемых систем Kafka может оказаться удобным наличие нескольких кластеров. Вот несколько причин этого.

- Разделение типов данных.
- Изоляция по требованиям безопасности.
- Несколько центров обработки данных (ЦОД) (восстановление в случае катаклизмов).

При работе, в частности, с несколькими ЦОД часто выдвигается требование копирования сообщений между ними. Таким образом, онлайн-приложения могут повсеместно получить доступ к информации о действиях пользователей. Например, если пользователь меняет общедоступную информацию в своем профиле, изменения должны быть видны вне зависимости от ЦОД, в котором отображаются результаты поиска. Или данные мониторинга могут собираться с многих сайтов в одно место, где расположены системы анализа и оповещения. Механизмы репликации в кластерах Kafka предназначены только для работы внутри одного кластера, репликация между несколькими кластерами не осуществляется.

Проект Kafka включает для этой цели утилиту *MirrorMaker*. По существу, это просто потребитель и производитель Kafka, связанные воедино очередью. Данная утилита получает сообщения из одного кластера Kafka и публикует их в другом. На рис. 1.8 демонстрируется пример использующей MirrorMaker архитектуры, в которой сообщения из двух локальных кластеров агрегируются в составной кластер, который затем копируется в другие ЦОД. Пускай простота этого приложения не создает у вас ложного впечатления о его возможностях по созданию сложных конвейеров данных, которые мы подробнее рассмотрим в главе 7.

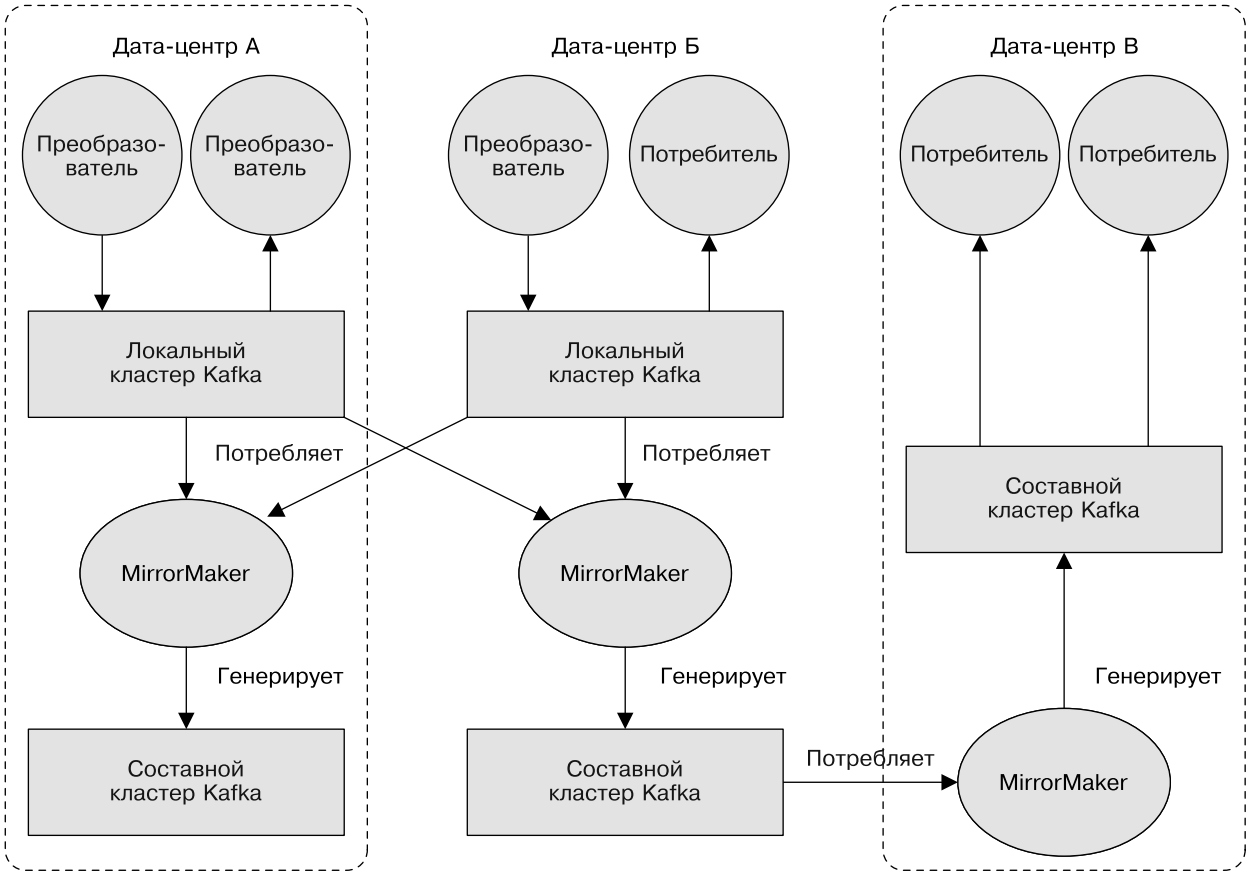

Рис. 1.8. Архитектура с несколькими ЦОД